SAE 1.0b (28.6.16)

Lecture Slides Quick Summary

1 Introduction

- Reasons why software is so hard to get right: complexity, change, competing objectives, constraints
- Software engineering is a collection of techniques, methodologies, and tools to help with above-mentioned challenges
- Overview of different parts of the lecture: omitted

2 Requirements Elicitation

2.1 Requirements

- Typical software development process: requirements elicitation, system design, detailed design, implementation, validation; often overlapping and iteratively
- A lot of IT projects fail because of lacking requirements
- **Definition: Requirement**: A feature that the system must have or a constraint it must satisfy to be accepted by the client.
- A requirement describes the user's view of the system; it describes the what and not the how
- There are different types of requirements, **functional** (functionality, external interfaces) and **non-functional** (performance, attributes/quality requirements, design constraints)
- Func. Req.: **Functionality** includes: relationship of outputs to inputs, response to abnormal situations, exact sequence of operations, validity checks on the input, effect of parameters
- Func. Req.: **External interfaces**: detailed description of all inputs and outputs
- Non-func. Req.: **performance**: static numerical requirements (e.g. amount of information handled) and dynamic numerical requirements (e.g. every query in less than two seconds)
- **Constraints** ("pseudo requirements"): standards compliance, implementation, operations, and legal requirements
- **Quality requirements** for criteria: correctness, completeness, consistency, clarity
- The sooner an error is found, the cheap to fix it
- As a QA step, requirements are validated (usually after elicitation or analysis) by reviews from clients and developers and prototyping

2.2 Activities

- **Activities**: identifying actors, identifying scenarios, identifying use cases, identifying nonfunctional requirements
- Actors represent roles (user type, external system, physical environment) – ask who needs to what
- Scenarios and use cases should document the behavior from the users' view and can be understood by them
- Scenario describes the common case and focuses on understandability
- Use case generalizes scenarios to describe all possible cases and focuses on completeness
- **Definition: scenario**: a narrative description of what people do and experience as they try to make use of computer systems and applications
- Scenarios have different applications throughout the software lifecycle
- **Sources of information**: users, client, existing documentation, task observation
- A **use case** consists of: unique name, initiating and participating actors, flow of events, entry and exit conditions, exceptions, and special requirements
- Nonfunctional requirements are defined together with function requirements because of their dependencies

- Typically, nonfunctional requirements conflict (e.g., Assembly/C for speed vs OO for maintainability)

3 Modeling and Specification

- To master complexity you must decompose it into smaller pieces.
- Decomposition allows for partitioning of development effort, supports unit testing + analysis, decouples parts, enables DRY
- After requirements elicitation + system design: **detailed design**
- Examples: permit null values, when to initialize fields, mutation: destructive updates or create a new object, concurrency/thread safety

3.1 Code Documentation

- Design decisions determine how code should be written – no matter at what stage of development (initial, extending, maintenance, ...)
- Source code documentation is insufficient
- Developers need documentation – which is difficult to extract from code

3.1.1 What to Document

- Essential properties: how to use the code/interface + how does the code work/implementation
- Docs should focus on what these properties are, not how they are achieved
- Method docs: parameters, return value, effects (heap, I/O, ...); justify assumptions
- Interface docs: global properties, evolution, abbreviations
- Key properties of docs: methods and constructors (prams, input state, effects, return), data structures (invariants), algorithms (justification, explanation, behavior of snippets)

3.1.2 How to Document

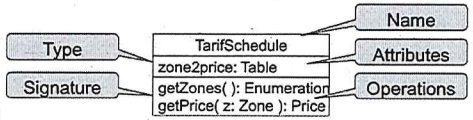
- Comments (Javadoc etc.; relies on conventions, limited tool support)
- Types and modifiers (static + runtime checking, auto-completion)
- Effect systems (e.g. throws IOException)
- Metadata/annotations (static checking with plugins, dynamic processing)
- Assertions (test case generation, runtime + static checking)
- Contracts (test case generation, runtime + static checking)
- Techniques are always a tradeoff between overhead, expensiveness, precision, and benefit
- Oftentimes a mix is best
- Better simplify than to describe complexity!

3.2 Informal Methods

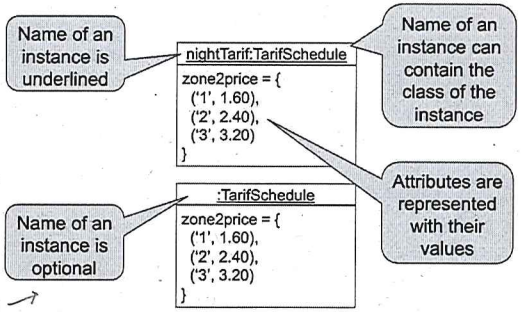
- **Underspecification**: software is typically designed iteratively, with each iteration adding details and reflecting on design decisions which were previously left open
- **Views**: different SE tasks require different perspectives: architecture, test data generation, security, deployment, ...
- Design specifications are models of the software providing suitable abstractions
- **Modeling** is building an abstraction of reality, which are simplifications – deals with complexity
- **UML** uses text + graphical notation – documents specs, analysis, design, and implementation; de facto standard
- UML notations: use case diagrams – requirements of a system; class diagrams – structure of a system; interaction diagrams – message passing (sequence + collaboration diagrams); implementation diagrams (component model – code dependencies; deployment model – structure of runtime system)

3.2.1 Static Models

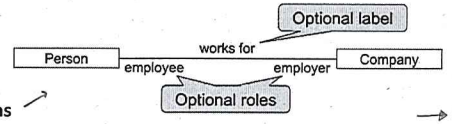
UML



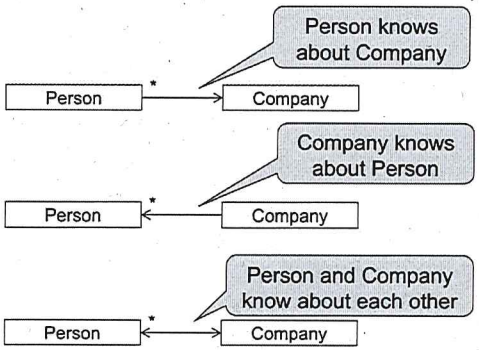
- **Classes**



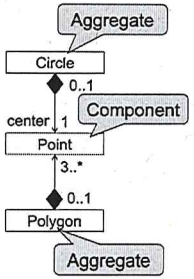
- **Instances/Objects**



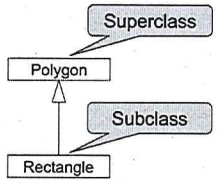
- **Associations** → can have **multiplicities** – specified above the line of the role; can be exact number, arbitrary, or range; associations can be directed



- A special form of association is **composition**, a "has-a" (exclusive part-of) relationship (no sharing)



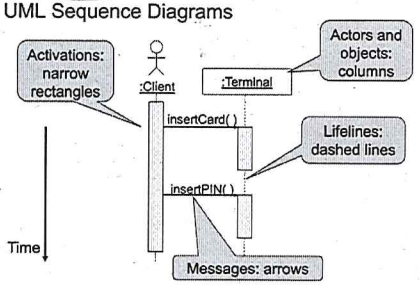
- Whereas **generalization** is a kind-of/is-a relationship, implemneted by inheritance



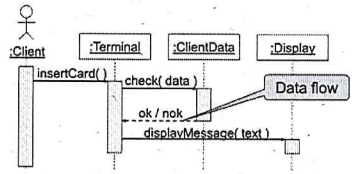
3.2.2 Dynamic Models

- Dynamic models describe the behavior of a systems – using sequence and state diagrams
- A UML **sequence diagram**

UML Sequence Diagrams

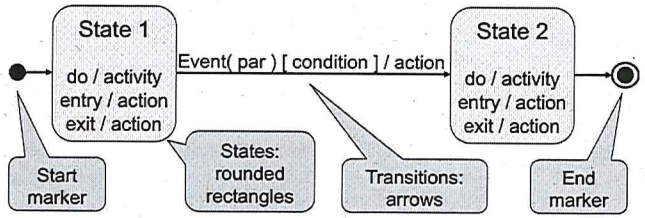


Nested Messages



- The source of an arrow indicates the activation which sent the message
- An activation is as long as all nested activations

- Creating an object is denoted by a message arrow pointing to the object, destruction is denoted by a cross (useful e.g. gc'ed envs)
- A **state** is an abstraction of the attribute values of an object – its an equivalence class of all those attributes values that do not need to be distinguished for the control structure of the class
- A UML **state diagram**



event condition action ECA

- Used for objects with extended lifespan
- An **event** is something happening at a point in time; an **action** is an operation in response to an event; an **activity** is an operation performed as long as the object is in a given state
- Construct dynamic models only for classes with significant dynamic behavior considering only the relevant attributes

3.2.3 Contracts

- Sometimes (e.g. marriage) diagrams are not enough as they do not capture the values of attributes
- **OCL (object constraint language)** is the contract language for UML
- A contract can mention: self, attributes, role names, side-effect free methods, logical connectives, operations on integers, reals, strings, sets, bags, sequences etc.
- It can use @pre to refer to the pre-state value

3.2.4 Mapping Models to Code

- Model-driven development: work on design models – generate code automatically
- Advantages: support many implementation platforms, no recurring tasks for devs, uniform code, can enforce coding conventions, models are no longer just docs
- Problems: abstraction mismatch (UML uses different abstractions than the programming language; how to map multiple inheritance), specifications may be incomplete and/or informal
- When automatically generated code if modified there has to be (complicated) syncing of models and code
- This works for basic properties, but interesting code is still written manually
- Other problems: maintain code without models (reverse-engineering); once code is modified, going back to the model is difficult/impossible
- Classes and inheritance are split into interfaces + implementation classes; attributes are non-public with gets and setters
- Associations are mapped to fields (or separate objects/collections)
- Synchronous messages in sequence diagrams are implemented by method calls
- State diagrams are implemented using switch-case

3.3 Formal Models

- Math-based and thus precise
- Enable automatic analysis
- Alloy is based on set theory

ALLOY

3.3.1 Static Models

Signatures

- A signature declares a set of atoms
  - Think of signatures as classes
  - Think of atoms as immutable objects
  - Different signatures declare disjoint sets

```
sig FSObject {}
```

- Extends-clauses declare subsets relations
  - File and Dir are disjoint subsets of FSObject

```
sig File extends FSObject {}
sig Dir extends FSObject {}
```

More on Signatures

- Signature can be abstract
  - Like abstract classes
  - Closed world assumption: the declared set contains exactly the elements of the declared subsets

```
abstract sig FSObject {}
sig File extends FSObject {}
sig Dir extends FSObject {}
```

```
FSObject = File + Dir
```

↳ assumption

- Signatures may constrain the cardinalities of the declared sets
  - 1 **one**: singleton set
  - 0..1 **lone**: singleton or empty set
  - 1+ **some**: non-empty set

```
one sig Root extends Dir {}
```

Operations on Sets

- Standard set operators
  - + (union)
  - & (intersection)
  - - (difference)
  - **in** (subset)
  - = (equality)
  - # (cardinality)
  - **none** (empty set)
  - **univ** (universal set)
- Comprehensions

```
sig File extends FSObject {}
sig Dir extends FSObject {}
```

```
#{ f: FSObject | f in File + Dir } >= #Dir
```

```
#( File + Dir ) >= #Dir
```

Operations on Relations

- Standard operators
  - -> (cross product)
  - . (relational join)
  - ~ (transposition)
  - ^ (transitive closure)
  - * (reflexive, transitive closure)
  - <: (domain restriction)
  - :> (range restriction)
  - ++ (override)
  - **iden** (identity relation)
  - [] (box join: e1[e2] = e2.e1)

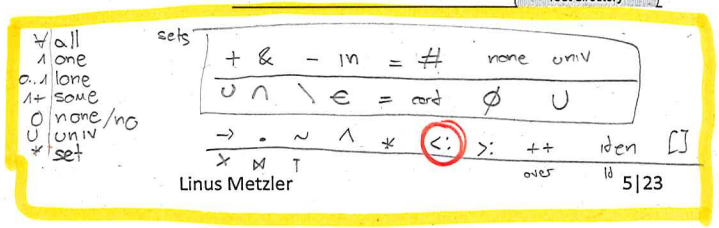
```
abstract sig FSObject {
  parent: lone Dir
}
```

```
sig Dir extends FSObject {
  contents: set FSObject
}
```

```
one sig Root extends Dir {}
```

```
FSObject in Root.^contents
```

All file system objects are contained in the root directory



Fields

- A field declares a relation on atoms
  - f is a binary relation with domain A and range given by expression e
  - Think of fields as associations

```
sig A {
  f: e
}
```

- Range expressions may denote multiplicities
  - **one**: singleton set (default)
  - **lone**: singleton or empty set
  - **some**: non-empty set
  - **set**: any set

```
abstract sig FSObject {
  parent: lone Dir
}
```

```
sig Dir extends FSObject {
  contents: set FSObject
}
```

Operations on Relations

- Standard operators
  - -> (cross product)
  - . (relational join)
  - ~ (transposition)
  - ^ (transitive closure)
  - * (reflexive, transitive closure)
  - <: (domain restriction)
  - :> (range restriction)
  - ++ (override)
  - **iden** (identity relation)
  - [] (box join: e1[e2] = e2.e1)

```
abstract sig FSObject {
  parent: lone Dir
}
```

```
sig Dir extends FSObject {
  contents: set FSObject
}
```

```
one sig Root extends Dir {}
```

```
FSObject in Root.^contents
```

Constraints

- Boolean operators
  - **!** or **not** (negation)
  - **&&** or **and** (conjunction)
  - **||** or **or** (disjunction)
  - **=>** or **implies** (implication)
  - **else** (alternative)
  - **<=>** or **iff** (equivalence)
- Four equivalent constraints

```
F => G else H
F implies G else H
(F && G) || ((!F) && H)
(F and G) or ((not F) and H)
```

- Quantified expressions
  - **some** e
    e has at least one tuple
  - **no** e
    e has no tuples
  - **lone** e
    e has at most one tuple
  - **one** e
    e has exactly one tuple

```
no Root.parent
```

Quantification

- Alloy supports five different quantifiers
  - ∀ **all** x: e | F
    F holds for every x in e
  - ∃ **some** x: e | F
    F holds for at least one x in e
  - **no** x: e | F
    F holds for no x in e
  - **lone** x: e | F
    F holds for at most one x in e
  - **one** x: e | F
    F holds for exactly one x in e
- Quantifiers may have the following forms
  - **all** x: e | F
  - **all** x: e1, y: e2 | F
  - **all** x, y: e | F
  - **all disj** x, y: e | F
- contents-relation is acyclic

```
no d: Dir | d in d.^contents
```

Predicates and Functions

- Predicates are named, parameterized formulas

```
pred p[ x1: e1, ..., xn: en ] { F }
```

```
pred isLeave[ f: FSObject ] {
  f in File || no f.contents
}
```

- Functions are named, parameterized expressions

```
fun f[ x1: e1, ..., xn: en ]: e { E }
```

```
fun leaves[ f: FSObject ]: set FSObject {
  { x: f.^contents | isLeave[ x ] }
}
```

Exploring the Model

- The Alloy Analyzer can search for structures that satisfy the constraints M in a model
- Find instance of a predicate
  - A solution to M &&
    some x1: e1, ..., xn: en | F

```
pred p[ x1: e1, ..., xn: en ] { F }
```

```
run p
```

- Find instance of a function
  - A solution to M &&
    some x1: e1, ..., xn: en,
    res: e | res = E

```
fun f[ x1: e1, ..., xn: en ]: e { E }
```

```
run f
```

- Missing or weak facts lead to **under-constrained** models (permitting undesired structures; easy to detect)
- Unnecessary facts **over-constrain** the model (excluding desired structures); in the extreme case, there are **inconsistencies** (e.g. assert 0 = 1, check that assertion)
- To avoid over-constraining, simulate the model (using run) and prefer assertions over facts

3.3.2 Dynamic Models

- Alloy has no built-in support for time or mutable state; has to be modeled explicitly
- It is easier to have operations only modify a global state (and move all relations + operations there)
- Alloy specifications are purely declarative, they abstract over irrelevant details
- Invariants in static models are facts, on dynamic models, they can be asserted as properties maintained by the operations
- Modeling temporal invariants: use **traces**; "util/ordering[Class]"
- Traces define a linear order on all states; first = initial state, subsequent states created by operating on the abstract state machine

3.3.3 Analyzing Models

- Validity and consistency checking for Alloy is undecidable; circumvented by checking validity + consistency within a given scope
- Internally Alloy represents everything as relations which in turn are a set of tuples; constraints and formulas are represented as formulas over relations
- And then it is processed by a SAT solver (NP-complete problem)
- A formula F is valid if it evaluates to true in *all* structures that satisfy the constraints C of the model
- Instead of enumerating all structures (possible but slow) within a given scope, Alloy looks for counterexamples
- **Summary**: check for consistency with run – positive answers are definite (structures); check for validity with check – negative answers are definite (counterexamples)
- Most interesting errors are found by looking at small instances

Consistency and Validity

- An Alloy model specifies a collection of constraints C that describe a set of structures
- Consistency:
  A formula F is consistent (satisfiable) if it evaluates to true in at least one of these structures

$$\exists s \bullet C(s) \wedge F(s)$$

- Validity:
  A formula F is valid if it evaluates to true in all of these structures

$$\forall s \bullet C(s) \Rightarrow F(s)$$

4 Modularity

4.1 Coupling

- Coupling measures interdependence between different modules – tighter coupling means its less easy to test, develop, change, understand, or reuse in isolation
- **Summary**: low coupling is desired, but comes with trade-offs: cohesion, performance + convenience, adaptability, code duplication; coupling to stable (e.g. library) classes Is less critical

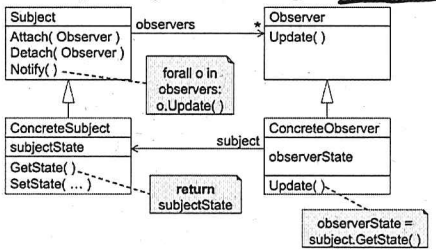
4.1.1 Data Coupling

- Modules exposing internal data representation become tightly coupled to their clients and thus difficult to change during maintenance, no strong invariants possible, concurrency is very complex; this can also lead to unexpected side effects
- One approach is to **restrict access to data/information hiding** + necessary checks: do not leak references to sub-objects, no capturing (storing arguments as sub-objects) and clone objects if necessary
- The **façade pattern** (or rather objects implementing it) provide a single, simplified interface to more general facilities of a module (without completely hiding the details)
- Another approach is **making shared data immutable** (avoids some problems of shared data, such as invariants, synchronization, side effects) but changing data representation remains a problem + copies can lead to overhead (time and space)
- The **flyweight pattern** maximizes sharing of immutable objects (Java uses it for Strings)
- Yet another approach is to **avoid shared data** at all but this can be cumbersome
- One style is the **pipe-and-filter style** where data flow is the only form of communication (i.e. no shared state) – components (filters) read data, compute, output and connectors (pipes) are streams (often async FIFO), possibly split-join; data is processed incrementally – thus filters must be independent (no shared state) and filters don't know of each other; e.g. Unix pipes
- Pipe-and-filter can be implemented in **StreamIt** which also supports split joins: split duplicate, split round-robin, join round-robin
- **Fusion** reduces communication cost at the expense of parallelism while **fission** is profitable if the benefits of parallelization outweigh the overhead introduced by fission
- **Advantages** of pipe-and-filter: reusability, easy to maintain, potential for parallelism

- **Weaknesses**: sharing global data is expensive or limiting, incremental filters are difficult to design, not appropriate for interactive applications, error handling is Achilles heel (some filter in the middle crashes), often the smallest common denominator in data processing (ASCII in Unix pipes)

4.1.2 Procedural Coupling

- Modules are couples to other modules whose methods they call
- Callers cannot be reused without callee modules
- When modules are procedurally coupled any change in the callees may requires changes in the caller
- One approach is to **move code**; this might reduce procedural coupling; sometimes functionality is duplicated to avoid dependences from other projects/companies
- Another approach is to use an **event-based style** using event emitters and event listeners (using callbacks); this might be done using the **observer pattern**

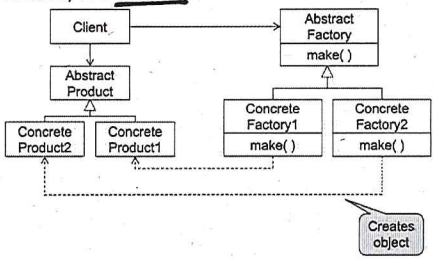


- Another approach is **MVC (model-view-controller architecture)**: models contain core functionality + data, views display information to the user, controllers handle user input; changes are propagated by events
- ⊕ **Strengths** of event-based style: strong support for reusability, adaption is done with minimum effect on other components
- ⊖ **Weaknesses** are loss of control (who will respond to an event and in what order) and ensuring correctness is difficult because it depends on the context in which it is invoked
- Another approach is to **restrict calls** by enforcing a policy restricting which modules may call what modules by e.g. using a layered architecture (presentation, logic, data)
- This layered approaching has bad performance but abstraction increases (partitioning complex problems), low coupling/easy maintenance, and reusability is good (different implementations of the same level can be exchanged)

4.1.3 Class Coupling

- Inheritance couples the subclass to the superclass – and changing the superclass may break the subclass
- Limiting options for inheritance relations is bound to cause issues (if even possible)
- One approach is to **rebalance inheritance with aggregation** – inheritance can be replaced by subtyping, aggregation, and delegation
- Another approach (to make it easier to change data structures during maintenance) is to **use interfaces**: replace class names by supertypes and use the most general supertype offering all required operations
- When allocating objects, coupling happens, too

- This leads to another approaching, **delegating allocations** by using dependency injection (→ reflection) and factories

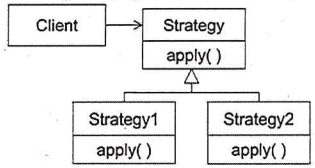


4.2 Adaption

- Changes – which often occur in software (perception of being easy to change) – often erode the structure of a system

4.2.1 Parametrization

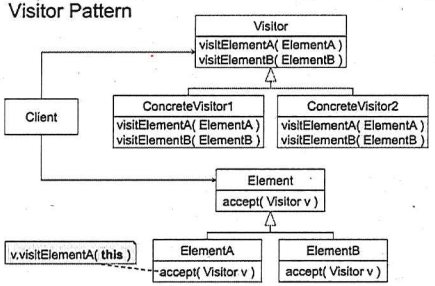
- Allow clients to customize the behavior by supplying different parameters
- Make modules parametric in: values, data structures, types, algorithms
- This can be achieved by using interfaces and factories (data structures), generic types (types), function objects (algorithms)



4.2.2 Specialization

- Allow clients to customize behavior by adding subclasses and overriding methods
- In OOP: overriding and dynamic method binding
- Dynamic method binding can also be used in case distinction (instanceof) – client code is adaptable and caller does not need to be changed; e.g. syntax tree
- Dynamic method binding also has drawbacks: reasoning (invariants), testing (more possible behaviors), versioning (harder to evolve without breaking subclasses), performance

Visitor Pattern

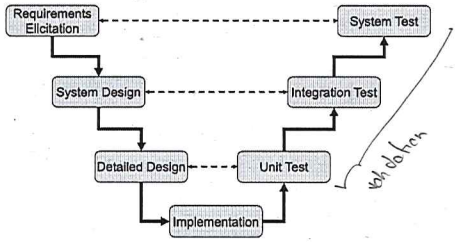


5 Testing

- Humans only have limited behavior to predict implementation behavior due to its complexity
- Humans make mistakes due to unclear requirements, wrong assumptions, design and coding errors

- Increase software reliability be fault avoidance (static analysis; development methodologies, reviews, verification), fault detection (dynamic analysis; testing) and fault tolerance (recover at runtime e.g. transactions, adding redundancy e.g. n-version programming)
- **A successful test finds errors**
- An error is a deviation of the observed behavior from the required/desired behaviors (functional requirements → user-acceptance testing; non-functional requirements → performance testing)
- Testing can only show the presence of bugs, not their absence
- It's impossible to completely test any nontrivial module or system (termination, time, cost)

5.1 Test Stages

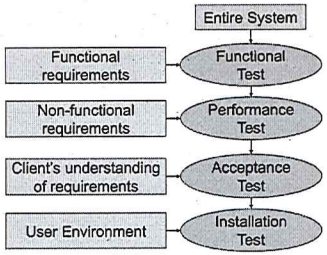


- A **test driver** applies test cases to the unit under test (UUT), including setup + teardown
- The **test stub** is a partial and temporary implementation of a component used by the UUT and simulates the activity if the missing component (answers to the UUT and returns fake data)
- **Unit testing** tests individual subsystems/collections of classes to confirm it's correctly coded and behaves as expected/intended
- To achieve reasonable coverage, every method has to be tested with several inputs (valid and invalid ones) – this is easier with **parametrized testing** (using values, ranges, and random values)
- Parametrized testing requires generic **testing oracles**
- Parametrized unit tests avoid boiler-place code but are difficult to write; especially useful when test data is generated automatically
- Test are re-executed after every change (automatically e.g. before every commit/push)
- Regression testing tests everything that used to work still works
- There are eight rules of testing:

1. Make sure all tests are fully automatic and check their own results
2. A test suite is a powerful bug detector that reduces the time it takes to find bugs
3. Run your tests frequently – every test at least once a day
4. When you get a bug report, start by writing a unit test that exposes the bug
5. Better to write and run incomplete tests than not run complete tests
6. Concentrate your tests on boundary conditions
7. Do not forget to test exceptions raised when things are expected to go wrong
8. Do not let the fear that testing can't catch all bugs stop you from writing tests that will catch most bugs

- Testing groups and eventually the entire system is called **integration testing** aiming to test interfaces between subsystems
- Typical strategies include: big-bang (non-incremental), bottom-up, top-down; selection criteria: amount of test harness (stubs + drivers), scheduling concerns

- **System testing** tests the entire system to determine whether it meets the requirements (func and non-func.)



- **Functional testing** treats the systems as a black box; test cases are designed from requirements specification, based on use case/user manual; tests describe input, event flow, output
- **Acceptance testing** is performed by the client; should show system meets the requirements and is ready to use – alpha (client at dev's site, controlled setting, dev fixing bugs) and beta testing (at client's site, no dev, realistic workout in target env)
- When performing ***independent* testing** one goal is to convince devs they do in fact make mistakes
- These testers must seek to break the software
- Independent testing is done when it comes to the system test (exception: acceptance test) and integration test
- Wrong conclusions about independent testing: devs shouldn't test, testers are only involved once the software is done, testers and devs shouldn't collaborate, testing team is responsible for QA

5.2 Test Strategies

- **Exhaustive testing** of all possible input is infeasible
- **Random testing** treats all inputs as equally valuable, avoids designer/tester bias (no boundary)
- **Functional testing** uses requirements knowledge; focus on input/output behavior, attempts to find missing/incorrect functions, interface and performance errors; doesn't effectively detect design and coding errors and doesn't reveal errors in the specifications
- **Structural testing** uses design knowledge about the system structure, algorithms, data structures to determine test cases, with the goal to cover all the code; not well suited for system test as it focuses on code and not on requirements (e.g. doesn't find missing logic); requires design knowledge (which testers + clients don't have and don't care about); thorough tests can be highly redundant

5.3 Functional Testing

5.3.1 Partition Testing

- Divide input into equivalence classes, with some classes having higher density of failure
- Choose test case for each equivalence class
- Examples: days in a month, leap years

5.3.2 Selecting Representative Values

- Select concrete values for each equivalence class from a range of valid values, also include below, within, and above the range; multiplicities on aggregations
- To select values from a discrete set, also include valid and invalid values; instances of each subclass
- A large number of errors tend to occur at boundaries of the input domain (overflow, comparisons, missing check for empty, wrong iteration number)
- Choose values at the edge of each equivalence class, limits, empty sets etc.

5.3.3 Combinatorial Testing

- Combining equivalence classes and boundary testing leads to many values for each input
- Reduce test cases to make effort feasible by using semantic constraints, combinatorial selection, and random selection
- Eliminate unnecessary conditions: especially for invalid values + use domain knowledge
- When selecting test data for objects consider object identities and aliasing
- Semantic constraints might decrease the number of test cases and increase coverage but still too many combinations, especially for e.g. object fields
- Empirical evidence suggest most errors don't depend on the interaction of many variables
- Instead of testing all possible combinations, focus on each pair of inputs; can be generalized to k-tuples
- Complexity of pairwise testing: n parameter, d values per parameter: number of test cases is logarithmic in n and quadratic in d; also holds for k $O(\log n + d^2)$
- Pairwise testing significantly reduces the number of test cases wile detecting most errors; is especially important when testing many system configurations; should be combined with other approaches to detect errors triggered by more complex parameter interaction

5.4 Structural Testing

- Detailed design and coding may introduce behaviors not present in the requirements (choice of data structures, algorithms, optimizations (e.g. caches))
- This is generally not exercised by functional testing e.g. AVL-tree rotation or cache misses

5.4.1 Control Flow Testing

- A **basic block** is a sequence of statements such that the code in a basic block: has *one entry point* (no code within it is the destination of a jump instruction) and has *one exit point* (only the last instruction causes the program to execute code in a different basic block); whenever the first instruction in a basic block is executed, the rest of the instructions are necessarily executed exactly once, in order
- An **intraprocedural control flow graph (CFG)** of a procedure p is a graph (N,E) where: N is the set of basic blocks in p plus designated entry and exit blocks; E contains an edge from a to b with condition c iff the execution of basic block a is succeeded by the execution of basic block b if condition c holds; an edge (entry, a, true) if a is the first basic block of p; edges (b, exit, true) for each basic block b that ends with an (possibly implicit) return statement
- A CFG can serves as an adequacy criterion for test cases – this is called **statement coverage** (executed statements divided by total statements); "a bug can only be detected if a statement is executed"
- 100% statement coverage doesn't guarantee anything
- **Branch coverage** (executed branches divided by total branches); an edge (m, n, c) in a CFG is a branch iff there is another edge (m, n', c') in the CFG with $n \neq n'$
- Branch coverage is more through than statement coverage (and a very widely used adequacy criterion) and complete branch coverage implies complete statement coverage; yet there are still problems
- To test all possible paths through the CFG, **path coverage** is used (executed paths divided by total paths); a path is a sequence of nodes n1, ..., nk such that: n1 = entry; nk = exit; there is an edge (ni, ni+1, c) in the CFG
- Due to unknown number of loop executions, an arbitrarily large number of test cases is needed for complete path coverage
- Path coverage testing is more thorough than statement and branch coverage (and complete path coverage implies completeness for the other two) but it's not feasible for loops

- **Loop coverage:** test each loop for zero, one, and more than one consecutive iterations (number of executed loops with 0,1,>1 iterations divided by total loops times three); typically combined with statement or branch coverage

5.4.2 Advanced Topics of Control Flow Testing

- Intraprocedural CFGs treat method calls as simple statement, yet statements may invoke different code depending on the **dynamic type of the receiver** – and testing should cover all possible behaviors (whose number explodes with dynamically-bound methods)
- To circumvent this huge number, use pairwise testing and use semantic constraints
- **Exceptions** add a control flow edge from where it is thrown to where it is caught; idea is to cover exceptional control flow like normal control flow during testing
- There is a difference between **checked** (invalid conditions outside the program's immediate control; invalid user input, DB, network, missing files etc.) and **unchecked** (present defects in the program or the execution environment; illegal arguments, null-pointer dereferencing, div by zero etc.)
- When testing the CFG, ignore checked exceptions, but consider throw statements
- Never use unchecked exceptions to control the flow
- When testing, do check checked exceptions, as they represent regular control flow

5.4.3 Data Flow Testing

- Since testing all paths is not feasible (exponential growth in the number of paths; loops), one idea is to test those paths where a computation on one part of the path affects the computation of another
- A **variable definition** for a variable v is a basic block that assigns to v; a **variable use** for a variable v is a basic block that reads the value from v
- A **definition-clear path** for a variable v is a path n1, ..., nk in the CFG such that: n1 is a variable definition for v; nk is a variable use for v; No ni ($1 < i \leq k$) is a variable definition for v (nk may be a variable definition if each assignment to v occurs after a use); note: definition-clear paths do not go from entry to exit
- A **definition-use (DU) pair** for a variable v is a pair of nodes (d,u) such that there is a definition-clear path d, ..., u in the CFG
- Now test all paths that provide a value for a variable use: **DU-pairs coverage** (executed DU-pairs divided by total DU-pairs)
- DU-pairs are computed using static **reaching-definitions** analysis:
- For each node n and for each variable v, compute all variable definitions for v that possibly reach n via a definition-clear path; the reaching definitions at a node n are: the reaching definitions of n's predecessors in the CFG, minus the definitions killed by one of n'd predecessors, plus the definitions made by one of n'd predecessors
- Reaching definitions algorithm:

Input
- pred(n) = { m | (m,n,c) is an edge in the CFG }
- succ(m) = { n | (m,n,c) is an edge in the CFG }
- gen(n) = { v_n | n is a variable definition for v }
- kill(n) = { v_m | n is a variable definition for v and $m \neq n$ }

We compute via fixpoint iteration
- Reach(n): The reaching definitions at the beginning of n
- ReachOut(n): The reaching definitions at the end of n

```
foreach node n do ReachOut( n ) := ∅ end
worklist := nodes
while worklist ≠ ∅ do
  n := any( worklist )
  remove n from worklist
  Reach( n ) := U_{m∈pred(n)} ReachOut( m )
  ReachOut( n ) := Reach( n ) \ kill( n ) ∪ gen( n )
  if ReachOut( n ) has changed then
    worklist := worklist ∪ succ( n )
  end
end
```

- To determine whether a definition + usage refer to the same heap structure, the static analysis would need arithmetic and aliasing information – instead it over-approximates

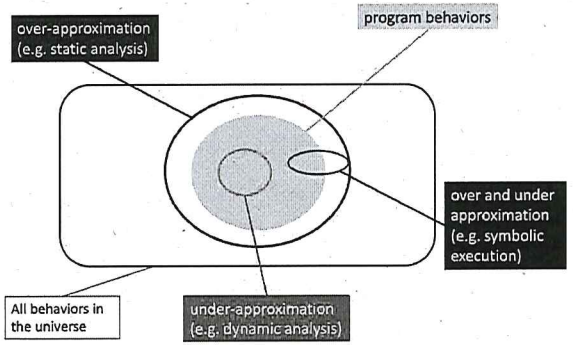
- Data flow testing complements control flow testing; not all pairs are feasible; there is more to dataflow testing than just DU-pairs; anomalies in DU-pairs may point to errors (use before define, double definition without use, termination after definition without use etc.)

5.4.4 Interpreting Coverage

- High coverage doesn't mean code is well tested – the contrary holds, however
- Coverage tools also identify which parts of the code haven't been tested
- **Experimental evaluation** (mutation testing) seeds defects into the code and checks how many of these defects were found i.e. how many tests (which passed before) failed
- This, of course, makes the test suite larger
- All adequacy criteria lead to test suites that detect more bugs than random testing, especially for large test suites

6 Introduction to Program Analysis

- It's *not* possible to build an automatic analyzer which takes as input an arbitrary program and an arbitrary property and then answer "yes" if the property certainly holds and "no" if the property certainly doesn't hold
- It's possible however, to build one which answers "no" if it is unknown whether the property holds



- Challenge: build an analyzer which answers "yes" as often as possible (because trivial solution is to answer "no" always)
- Static analyzers can automatically compute invariants per program point

6.1 Abstract Interpretation

- A general theory on how to approximate systematically; relate the concrete with the abstract, finite with infinite; many analyses can be seen as abstract interpreters
- Step-by-step:

1. Select/define abstract domain (based on the properties you want to prove)
2. Define abstract semantics for the language w.r.t. to the domain (sound w.r.t. concrete semantics; define abstract transforms → effect of statement/expression on the abstract domain)
3. Iterate abstract transformers over the abstract domain; until you reach a **fixed point** which is the **over-approximation** of the program

- Over-approximation reduces the space we need to enumerate
- An **abstract program state** is a map from variables to elements in the domain
- An **abstract transformer** describes the effect of a statement and expression evaluation on an abstract state

- Abstract transformers are defined per programming language, and *not* per program – thus they essentially define the new abstract semantics of the language; any program in the same language can use the same transformers
- A correct abstract transformer should always produce results that are a **superset** of what a concrete transformer would produce
- To be sound and *im*precise: output T; yet sound and precise is desirable – when losing precision, it should be clear why and where: sometimes computing the best transformer is impossible and efficiency and precision can be conflicting goals
- Abstract domains: sign, interval
- To produce the least upper bound of two elements A and B, you can **join** $\sqcup$ them; we have $A \sqsubseteq A \sqcup B$ and $B \sqsubseteq A \sqcup B$; $D \sqsubseteq E$ means E is *more abstract* than D
- When a domain has infinite height (e.g. intervals) it may not be possible to reach a fixed point – use **widening** (which is at the *expense of precision*) (e.g. go to ∞)

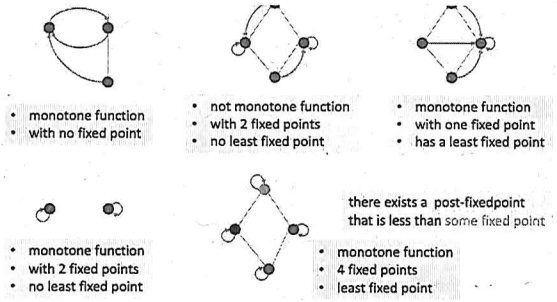
7 Math Background for Analysis

7.1 Structures

- A **partial order** is a binary relation $\sqsubseteq \subseteq L \times L$ on a set L with these properties: reflexive $\forall p \in L: p \sqsubseteq p$, transitive $\forall p, q, r \in L: (p \sqsubseteq q \wedge q \sqsubseteq r) \Rightarrow p \sqsubseteq r$, anti-symmetric $\forall p, q \in L: (p \sqsubseteq q \wedge q \sqsubseteq p) \Rightarrow p = q$
- A poset $(L, \sqsubseteq)$ is a set L equipped with a partial ordering $\sqsubseteq$, e.g. $(\wp(L), \subseteq)$ is a poset with the power set
- Intuition of posets: capture implication between facts - $p \sqsubseteq q$ intuitively means $p \Rightarrow q$; also p is "more precise" than q i.e. p represents fewer concrete states than q
- Given a poset $(L, \sqsubseteq)$, an element $\bot \in L$ is called the **least element** if it is small than all other elements of the poset: $\forall p \in L: \bot \sqsubseteq p$. The **greatest element** is an element T if $\forall p \in L: p \sqsubseteq \mathrm{T}$. The least and greatest elements may not exist, but if they do, they are unique
- Given a poset $(L, \sqsubseteq)$ and $Y \subseteq L$: $u \in L$ is an **upper bound** of Y if $\forall p \in Y: p \sqsubseteq u$; $l \in L$ is a **lower bound** of Y if $\forall p \in Y: p \sqsupseteq l$
- These bounds for Y *may not exist*
- $\sqcup Y \in L$ is a **least upper bound** of Y is $\sqcup Y$ is an upper bound of Y and $\sqcup Y \sqsubseteq u$ whenever u is another upper bound of Y
- $\sqcap Y \in L$ is a **greatest lower bound** of Y is $\sqcap Y$ is a lower bound of Y and $\sqcap Y \sqsupseteq l$ whenever l is another lower bound of Y
- $\sqcup Y$ and $\sqcap Y$ don't have to be in Y
- Abbreviate: $p \sqcup q$ for $\sqcup \{p, q\}$ and $p \sqcap q$ for $\sqcap \{p, q\}$
- A **complete lattice** $(L, \sqsubseteq, \sqcup)$ is a poset where $\sqcup Y$ and $\sqcap Y$ exist for any $Y \subseteq L$, e.g. $(\wp(L), \subseteq, \cup, \cap)$; sign domain, interval domain, the set of traces $[\![P]\!]$

7.2 Functions

- A function $f: A \to B$ between two posets $(A, \sqsubseteq)$ and $(B, \leq)$ is **increasing (monotone)**: $\forall a, b \in A: a \sqsubseteq b \Rightarrow f(a) \leq f(b)$; oftentimes a special case is used, where the function is between elements in the same poset: $\forall a, b \in A: a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$
- For a poset $(L, \sqsubseteq)$, function $f: L \to L$, element $x \in L$: x is a **fixed point** iff $f(x) = x$; x is a **post-fixedpoint** iff $f(x) \sqsubseteq x$; $\mathrm{Fix}(f)$ is the set of all fixed points, $\mathrm{Red}(f)$ the set of all post-fixed points
- For a poset $(L, \sqsubseteq)$, a function $f: L \to L$, we say that $lfp \sqsubseteq f \in L$ is a **least fixed point** of f if: $(lfp \sqsubseteq f)$ is a fixed point and it is the least fixed point: $\forall a \in L: a = f(a) \Rightarrow (lfp \sqsubseteq f) \sqsubseteq a$; the least fixed point may not exist



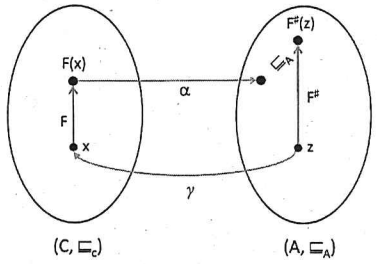
- **Tarski's fixed point theorem**: if $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \mathrm{T})$ is a complete lattice and $f: L \to L$ is a monotone function, then $lfp \sqsubseteq f$ exists and $lfp \sqsubseteq f = \sqcap \mathrm{Red}(f) \in \mathrm{Fix}(f)$; note the complete lattice can be of infinite height
- For a poset $(L, \sqsubseteq)$, function $f: L \to L$, element $a \in L$, the **iterates of the function** from a are: $f^0(a), f^1(a), f^2(a), \ldots$ where $f^{n+1}(a) = f(f^n(a))$ and $f^0(a) = a$; in program analysis, $a = \bot$
- Given a poset of finite height, a least element $\bot$, monotone f: the iterates $f^0(\bot), f^1(\bot), f^2(\bot), \ldots$ for an increasing sequence which eventually stabilizes from some $n \in$, i.e.: $f^n(\bot) = f^{n+1}(\bot)$ and $(lfp \sqsubseteq f) = f^n(\bot)$ - thus $lfp \sqsubseteq f$ can be computed by a simple algorithm

7.3 Approximating Functions

- Let $[\![P]\!]$ be the set of reachable states of a program P. Also let the function F be $F(S) = I \cup \{c' | c \in S \wedge c \to c'\}$ where I is an initial set of states and $\to$ is the transition relation between states. Then $[\![P]\!]$ is a **fixed point** of F: $F([\![P]\!]) = [\![P]\!]$ (in fact, $[\![P]\!]$ is the *least* fixed point of F)
- Static Program Analysis:

1. Define a function $F^\#$ st $F^\#$ approximates F; this is done once and for all per programming language
2. The least fixed point of $F^\#$, some V, approximates the last fixed point of F, some $[\![P]\!]$
3. Automatically compute a fixed point of $F^\#$ i.e. a V where $F^\#(V) = V$

- Given $F: C \to C, F^\#: C \to C$ an approximation $F^\#$ of F means $\forall x \in C: F(x) \sqsubseteq_C F^\#(x)$
- Given $F: C \to C, F^\#: A \to A$ we need to connect the concrete C and the abstract A via two functions: $\alpha: C \to A$, the abstraction function, and $\gamma: A \to C$, the concretization function
- **Definition 1**: given $F: C \to C, F^\#: A \to A$ and we know α, γ form a Galois connection[1], then $\forall z \in A: \alpha(F(\gamma(z))) \sqsubseteq_A F^\#(z)$. What this equation says, is the following: if we have some function in the abstract that we think should approximate the concrete function, then to check that this is indeed true, we need to prove that for any abstract element, concretizing it, applying the concrete function and abstracting back again is less than applying the function in the abstract directly.

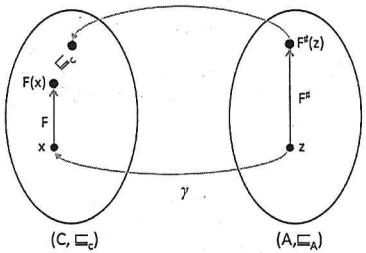


[1] Not relevant; places some restrictions on what α, γ can be e.g. α has to be monotone

- The *least precise* approximation of F is always $F^{\#}(z) = \mathrm{T}$ which is always sound, given by $\forall z \in A: \alpha\left(F(\gamma(z))\right) \sqsubseteq_A \mathrm{T}$
- The *best abstract function* is $F^{\#}(z) = \alpha\left(F(\gamma(z))\right)$; often however, algorithmically not implementable, yet there is an $F^{\#}$ which has the same behavior but a different implementation; any such $F^{\#}(z)$ is called the **best transformer**
- **Key theorem I** Given the following:
  1. Monotone functions $F: C \to C, F^{\#}: A \to A$
  2. $\alpha: C \to A, \gamma: A \to C$ forming a Galois connection
  3. $\forall z \in A: \alpha\left(F(\gamma(z))\right) \sqsubseteq_A F^{\#}(z)$ (i.e. $F^{\#}$ approximates F)

  THEN $\alpha(lfp(F)) \sqsubseteq_A lfp(F^{\#})$

  This is important as it goes from local function approximation to global approximation. This is a key theorem in program analysis
- Premises usually proved manually; least fixed point in the abstract computed automatically and also sound
- **Definition 2**: given $F: C \to C, F^{\#}: A \to A$ and we know α, γ *do not* form a Galois connection, then $\forall z \in A: F(\gamma(z)) \sqsubseteq_C \gamma\left(F^{\#}(z)\right)$.



- **Key theorem II** Given the following:
  1. Monotone functions $F: C \to C, F^{\#}: A \to A$
  2. $\gamma: A \to C$ is monotone
  3. $\forall z \in A: F(\gamma(z)) \sqsubseteq_C \gamma\left(F^{\#}(z)\right)$ (i.e. $F^{\#}$ approximates F)

  THEN $lfp(F) \sqsubseteq_A \gamma\left(lfp(F^{\#})\right)$
- $F^{\#}$ is to be defined for the particular abstract domain A that we work with. The domain A can be Sign, Parity, Interval, Octagon, Polyhedra, and so on. In our setting and commonly, we simply keep a map from every label (program counter) in the program to an abstract element in A Then $F^{\#}$ simply updates the mapping from labels to abstract elements
- $F^{\#}: (\mathrm{Label} \to \mathrm{A}) \to (\mathrm{Label} \to \mathrm{A})$
- $F^{\#}(m)\ell = \begin{cases} \top & \text{if } \ell \text{ is initial label} \\ \bigsqcup_{(\ell', \mathrm{action}, \ell)} [\![\mathrm{action}]\!](m(\ell')) & \text{otherwise} \end{cases}$
- An action can be $b \in \mathrm{BExp}, x := a \in \mathrm{AExp}, \mathrm{skip}$; $[\![\mathrm{action}]\!]$ captures the abstract semantics of the language for a particular abstract domain
- In performing an action, the assignment and the boolean expression of a conditional is fully evaluated
- The Interval domain is an example of a **non-relational domain**. It does not explicitly keep the relationship between variables. In some cases however, it may be necessary to keep this relationship in order to be more precise. Next, we show two examples of abstractions (Octagon and Polyhedra) where the relationship is kept. These domains are called **relational domains**
- **Octagon**: fixed slope; the abstract state is a map from labels to conjunction of constraints; form of constraints: $\pm x \pm y \leq c$
- **Polyhedra**: variable slope; the abstract state is a map from labels to conjunction of constraints; form of constraints: $c_1 x_1 + c_2 x_2 + \cdots + c_n x_n \leq c$

cheat sheet

8 Interval Analysis

Mathematical Concept	Use in Static Analysis
Complete Lattice	Defines Abstract Domain and ensure joins exist.
Joins (⊔)	Combines facts arriving at a program point
Bottom (⊥)	Used for initialization of all but initial elements
Top (T)	Used for initialization of initial elements
Widening (▽)	Used to guarantee analysis termination
Function Approximation	Critical to make sure abstract semantics approximate the concrete semantics
Fixed Points	This is what is computed by the analysis
Tarski's Theorem	Ensures fixed points exist.

- Our **starting point** is a domain where each element of the domain is a set of states. The domain of states is a complete lattice: $(\wp(\Sigma), \subseteq, \cup, \cap, \emptyset, \Sigma), \Sigma = \mathrm{Lab} \times \mathrm{Store}$

8.1 Abstract Interpretation: Step 1- Select/Define an Abstract Domain

- **Interval domain:** If we are interested in properties that involve the range of values that a variable can take, we can abstract the set of states into a map which captures the range of values that a variable can take

Interval Domain: Lets Define it

Let the interval domain be: $(\mathbb{L}^{\mathbb{I}}, \sqsubseteq_{\mathbb{I}}, \sqcup_{\mathbb{I}}, \sqcap_{\mathbb{I}}, \perp_{\mathbb{I}}, [-\infty,\infty])$

We denote $\mathbb{Z}^{\infty} = \mathbb{Z} \cup \{-\infty, \infty\}$
The set $\mathbb{L}^{\mathbb{I}} = \{[x,y] \mid x,y \in \mathbb{Z}^{\infty},\ x \leq y\} \cup \{\perp_{\mathbb{I}}\}$

For a set $S \subseteq \mathbb{Z}^{\infty}$, `min(S)` returns the minimum number in `S`, `max(S)` returns the maximum number in `S`.

- $[a,b] \sqsubseteq_{\mathbb{I}} [c,d]$ if $c \leq a$ and $b \leq d$
- $[a,b] \sqcup_{\mathbb{I}} [c,d] = [\min(\{a,c\}), \max(\{b,d\})]$
- $[a,b] \sqcap_{\mathbb{I}} [c,d] = [\mathrm{meet}(\max(\{a,c\}), \min(\{b,d\}))]$
  where `meet(a,b)` returns `[a,b]` if $a \leq b$ and $\perp_{\mathbb{I}}$ otherwise

Intervals: Applied to Programs

The $\mathbb{L}^{\mathbb{I}}$ domain simply defines intervals, but to apply it to programs we need to take into account program labels (program counters) and program variables.

Therefore, for programs, we use the domain $\mathtt{Lab} \to (\mathtt{Var} \to \mathbb{L}^{\mathbb{I}})$

That is, at each label and for each variable, we will keep the range for that variable. This domain is also a complete lattice.

The operators of $\mathbb{L}^{\mathbb{I}}$ $\sqsubseteq_{\mathbb{I}}, \sqcup_{\mathbb{I}}, \sqcap_{\mathbb{I}}$ are lifted directly to both domains:
- $\mathtt{Var} \to \mathbb{L}^{\mathbb{I}}$
- $\mathtt{Lab} \to (\mathtt{Var} \to \mathbb{L}^{\mathbb{I}})$

Intervals: Applied to Programs

$\alpha^{\mathbb{I}}: \wp(\Sigma) \to (\mathrm{Lab} \to (\mathrm{Var} \to \mathbb{L}^{\mathbb{I}}))$
$\gamma^{\mathbb{I}}: (\mathrm{Lab} \to (\mathrm{Var} \to \mathbb{L}^{\mathbb{I}})) \to \wp(\Sigma)$

Using $\alpha^{\mathbb{I}}$, we abstract a set of states into a map from program labels to interval ranges for each variable.

Using $\gamma^{\mathbb{I}}$, we concretize the intervals maps to a set of states

Formal definition of $\alpha^{\mathbb{I}}$ and $\gamma^{\mathbb{I}}$ is left as an exercise.

Example of Abstraction and Concretization

$\alpha^{\mathbb{I}}$ (
{ (1,{x↦1,y↦9,q↦-2}), (1,{x↦5,y↦9,q↦-2}), (1,{x↦8,y↦9,q↦-2}),
(1,{x↦1,y↦9,q↦4}), (1,{x↦5,y↦9,q↦4}), (1,{x↦8,y↦9,q↦4}) }
)
= 1 ↦ (x ↦ [1,8], y ↦ [9,9], q ↦ [-2,4])

$\gamma^{\mathbb{I}}$ (1 ↦ (x ↦ [1,8], y ↦ [9,9], q ↦ [-2,4]))

= { (1,{x↦1,y↦9,q↦-2}), (1,{x↦5,y↦9,q↦-2}), (1,{x↦8,y↦9,q↦-2}),
(1,{x↦1,y↦9,q↦4}), (1,{x↦5,y↦9,q↦4}), (1,{x↦8,y↦9,q↦4}),
(1,{x↦7,y↦9,q↦3}), (1,{x↦3,y↦9,q↦4}), (1,{x↦1,y↦9,q↦-1}),
…, …, … }

8.2 Abstract Interpretation: Step 2- Define Abstract Semantics for The Language

- Desired is $F^i: \big(\text{Lab} \to (\text{Var} \to \text{L}^i)\big) \to \big(\text{Lab} \to (\text{Var} \to \text{L}^i)\big)$ such that $\alpha^i(lfp(F)) \sqsubseteq lfp(F^i)$, the best transformer is $\alpha^i \circ F \circ \gamma^i$
- $F^i(m)\ell = \begin{cases} \lambda v.\,[-\infty,\infty] & \text{if } \ell \text{ is initial label} \\ \bigsqcup_{(\ell',\text{action},\ell)} [\![\text{action}]\!]_i(m(\ell')) & \text{otherwise} \end{cases}$ $[\![\text{action}]\!]_i: (\text{Var} \to \text{L}^i) \to (\text{Var} \to \text{L}^i)$
- $(\ell', \text{action}, \ell)$ is an edge in the **control-flow graph**; formally speaking, if there exists a transition $t = \langle \ell', \sigma' \rangle \to \langle \ell, \sigma \rangle$ in a program trace in P, where t was performed by statement called action, then $(\ell', \text{action}, \ell)$ must exist. This says that we are sound: we never miss a flow
- However, $(\ell', \text{action}, \ell)$ may exist even if no such transition t above occurs. In this case, the analysis would be imprecise as we would unnecessarily create more flows
- An action can be $b \in \text{BExp}, \text{x} := a \in \text{AExp}, \text{skip}$; $[\![\text{action}]\!]$ should produce sound and precise results
- $[\![x := a]\!]_i(m) = m[x \mapsto v]$, where $\langle a, m \rangle \Downarrow_i v$; $\langle a, m \rangle \Downarrow_i v$ says that given a map m, the expression a evaluates to a value $v \in L^i$
- The operational semantics rules for expression evaluation are as before, except: any constant Z is abstracted to an element in L^i and operators +, - and * are re-defined for the Interval domain
- $[l_1, u_1] \leq [l_2, u_2] = ([l_1, u_1] \sqcap_i [-\infty, u_2], [l_1, \infty] \sqcap_i [l_2, u_2])$

8.3 Abstract Interpretation: Step 3- Iterate Abstract Transformers Over Abstr. Domain

- Generally, if we have a complete lattice $(L, \sqsubseteq, \sqcup, \sqcap)$ and a monotone function F, if the height is infinite or the computation of the iterates of F takes too long, we need to find a way to approximate the least fixed point of F. The interval domain and its function F^i is an example of this case. We need to find a way to compute an element A such that: $lfp^{\sqsubseteq}F \sqsubseteq A$
- The operator $\nabla: L \times L \to L$ is called **widening operator** if
  → $\forall a, b \in L: a \sqcup b \sqsubseteq a \nabla b$ (widening approximates join)
  → if $x^0 \sqsubseteq x^1 \sqsubseteq x^2 \sqsubseteq \cdots \sqsubseteq x^n \sqsubseteq \cdots$ is an increasing sequenece then $y^0 \sqsubseteq y^1 \sqsubseteq y^2 \sqsubseteq \cdots \sqsubseteq y^n$ **stabilizes** after a **finite number of steps**
  where $y^0 = x^0$ and $\forall i \geq 0: y^{i+1} = y^i \nabla x^{i+1}$
- Widening is completely independent of the function F; its defined (like join) for a particular domain

Useful Theorem

If L is a complete lattice, $\nabla: \text{L} \times \text{L} \to \text{L}$, F: L → L is monotone
Then the sequence:

$y^0 = \bot$
$y^1 = y^0 \nabla F(y^0)$
$y^2 = y^1 \nabla F(y^1)$
...
$y^n = y^{n-1} \nabla F(y^{n-1})$

will stabilizes after a finite number of steps with y^n being a post-fixedpoint of F.

By Tarski's theorem, we know that a post-fixedpoint is above the least fixed point. Hence, it follows that: $lfp^{\sqsubseteq}F \sqsubseteq y^n$

Widening for Interval Domain

Let us define a widening operator for the intervals

$$\nabla_i: \text{L}^i \times \text{L}^i \to \text{L}^i$$

[a, b] ∇_i [c, d] = [e, f] where:
if c < a, then e = -∞, else e = a
if d > b, then f = ∞, else f = b

if one of the operands is ⊥ the result is the other operand.
The basic intuition is that if we see that an end point is unstable, we move its value to the extreme case.

Exercise: show this operator satisfies the conditions for widening.

Chaotic (Asynchronous) Iteration

```
x1 := ⊥; x2 = ⊥; ...; xn = ⊥;
W := {1,...,n};

while (W ≠ {}) do {
  ℓ := removeLabel(W);
  prevℓ := xℓ;
  xℓ := prevℓ ∇ fℓ(x1,...,xn);
  if (xℓ ≠ prevℓ)
    W := W ∪ influence(ℓ);
}
```

- W is the worklist , a set of labels left to be processed
- influence(ℓ) returns the set of labels where the value at those labels is influenced by the result at ℓ
- Re-compute only when necessary, thanks to influence (ℓ)
- Asynchronous computation can be parallelized

9 Pointer Analysis

TO DO

10 Symbolic Execution

10.1 Symbolic Execution

- Symbolic execution is a technique which sits in between testing and static analysis. It is completely automatic, and aims to explore as many program executions as possible, with the expense that it has false negatives: it may miss program executions, that is, may miss errors. Hence, symbolic execution is a particular instance of an under-approximation (some versions are actually both under- and over- approximations)
- Symbolic execution is widely used in practice. Tools based on symbolic execution have found serious errors and security vulnerabilities in various systems
- In classic symbolic execution, we associate with each variable a symbolic value instead of a concrete value. We then run the program with the symbolic values obtaining a big constraint formula as we run the program. Hence, the name symbolic execution.
- At any program point we can invoke a constraint (SMT) solver to find satisfying assignments to the formula. These satisfying assignments can be used to indicate real concrete inputs for which the program reaches a program point or to steer the analysis to another part of the program.
- At any point during program execution, symbolic execution keeps two formulas: **symbolic store** and a **path constraint**

- Therefore, at any point in time the symbolic state is described as the conjunction of these two formulas

Symbolic Store

- The values of variables at any moment in time are given by a function $\sigma_s \in \text{SymStore} = \text{Var} \rightarrow \text{Sym}$
  - Var is the set of variables as before
  - Sym is a set of symbolic values
  - σ_s is called a symbolic store
- Example: $\sigma_s : \texttt{x} \mapsto \texttt{x0}, \ \texttt{y} \mapsto \texttt{y0}$

Semantics

- Arithmetic expression evaluation simply manipulates the symbolic values
- Let $\sigma_s : \texttt{x} \mapsto \texttt{x0}, \ \texttt{y} \mapsto \texttt{y0}$
- Then, `z = x + y` will produce the symbolic store: $\texttt{x} \mapsto \texttt{x0}, \ \texttt{y} \mapsto \texttt{y0}, \ \texttt{z} \mapsto \texttt{x0+y0}$

That is, we literally keep the symbolic expression `x0+y0`

- The analysis keeps a **path constraint** (pct) which records the history of all branches taken so far.
  The path constraint is simply a formula
  The formula is typically in a decidable logical fragment without quantifiers
  At the start of the analysis, the path constraint is true
  Evaluation of conditionals affects the path constraint, but not the symbolic store
- A serious limitation of symbolic execution is handling unbounded loops.
- Symbolic execution runs the program for a finite number of paths. But what if we do not know the bound on a loop? The symbolic execution will keep running forever
- A common solution in practice is to provide some loop bound. In this example, we can bound k, to say 2. This is an example of an under approximation. Practical symbolic analyzers usually under-approximate as most programs have unknown loop bounds.
- Constraint solving is fundamental to symbolic execution as a constraint solver is continuously invoked during analysis. Often, the main roadblock to performance of symbolic execution engines is the time spent in constraint solving. Therefore, it is important that:
  1. The SMT solver supports as many decidable logical fragments as possible. Some tools use more than one SMT solver.
  2. The SMT solver can solve large formulas quickly.
  3. The symbolic execution engines tries to reduce the burden in calling the SMT solver by exploring domain specific insights.
- The basic insight here is that oftentimes the analysis will invoke the SMT solver with similar formulas. Therefore, the symbolic execution system can keep a map (cache) of formulas to a satisfying assignment for the formula
- Then, when the engine builds a new formula and would like to find a satisfying assignment for that formula, it can first access the cache, before calling the SMT solver
- One key optimization is **caching**: if we get a weaker formula than one we've already solved, we can
- immediately reuse the solution already found in the cache, without calling the SMT solver; if we get a stronger formula as a query, then we can quickly try the solution in the cache and see if it works, without calling the solver
- Since SMTs don't handle non-linear constraints well, an alternative is concolic execution to under-approximate these constraints

10.2 Concolic Execution

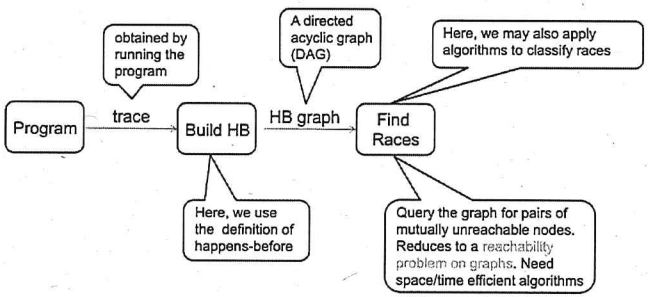
- Concolic Execution: combines both symbolic execution and concrete (normal) execution
- The basic idea is to have the concrete execution drive the symbolic execution

- Here, the program runs as usual (it needs to be given some input), but in addition it also maintains the usual symbolic information

11 Race Detection

- Dynamic race detecting is a popular kind of dynamic analysis; it's an under approximation, highly effective for finding concurrency bugs; tradeoff between asymptotic complexity and precision
- *Semantically*, a **data race** occurs when we have a reachable program state where: we have two outgoing transitions by two different threads; the two threads access the same memory location; one of the accesses is a write
- The definition of a data race suggests a naïve algorithm which finds all races of a program given some input states. The algorithm simply enumerates all reachable states of the concurrent program from the initial input states and checks the definition on each such reachable state. *This is not feasible in practice*
- In practice, algorithms aim to scale to large programs by being more efficient and not keeping program states around. To accomplish that, they weaken their guarantees
- A typical guarantee is that the first race the algorithm repots is a real race, but any subsequent reported races after the first race are not guaranteed to exist, that is, they may be false positives, a major issue to deal with for any modern analyzer.
- **Modern Dynamic Race Detection:**

1. Define Memory locations (on which races can happen); usually easy but there can be issues (framework vs. user-code)
2. Define Happens-Before Model (how operations are ordered); can be tricky to get right due to subtleties of concurrency
3. Come up with an Algorithm to detect races; hard to get good asymptotic complexity + correctness
4. Come up with techniques (algorithm, filters) to remove harmless races; needs to answer what harmless means
5. Implement Algorithm and Evaluate; important to have low instrumentation overhead


obtained by running the program
A directed acyclic graph (DAG)
Here, we may also apply algorithms to classify races
Program
trace
Build HB
HB graph
Find Races
Here, we use the definition of happens-before
Query the graph for pairs of mutually unreachable nodes. Reduces to a reachability problem on graphs. Need space/time efficient algorithms

- Example domain: event-driven applications
- **No false negatives:** if the Analysis reports no races on an execution, then the execution must not contain a race
- **No false positives:** if the Analysis reports a race for a given execution then the execution for sure contains a race
- Synchronization done with reads/writes leads to thousands of false races; prevented by race coverage
- Massive number of event handler quickly causes space blow-up in analysis data structure; prevented by using greedy chain decomposition + vector clocks

- A race detector should compute races. The basic query is whether two operations a and b are ordered: $a \preccurlyeq b$; observation: represent $\preccurlyeq$ (the happens-before of an execution trace) as a directed acyclic graph and perform graph connectivity queries to answer $a \preccurlyeq b$; report a race if a and b are not reachable from one another, they reach the same memory location and one is a write. (not ordered)
- When combining chain decomposition with vector clocks, the space and time complexity is manageable – in the optimal version with $C, C \ll N$ chains, the chain computation time is $\mathcal{O}(C \cdot M)$, vector clock computation takes $\mathcal{O}(C \cdot M)$, query is $\mathcal{O}(1)$, and space $\mathcal{O}(C \cdot N)$